Army Research Laboratory

A Plant Model for Smart Projectiles

by Robert J. Yager

ARL-TR-5520 April 2011

Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5066

ARL-TR-5520 **April 2011**

A Plant Model for Smart Projectiles

Robert J. Yager
Weapons and Materials Research Directorate, ARL

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*		

1. REPORT DATE *(DD-MM-YYYY)* April 2011	**2. REPORT TYPE** Final	**3. DATES COVERED (From - To)** 1 May 2009–16 February 2011
4. TITLE AND SUBTITLE A Plant Model for Smart Projectiles		**5a. CONTRACT NUMBER**
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Robert J. Yager		**5d. PROJECT NUMBER** AH80
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) U.S. Army Research Laboratory ATTN: RDRL-WML-A Aberdeen Proving Ground, MD 21005-5066		**8. PERFORMING ORGANIZATION REPORT NUMBER** ARL-TR-5520
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		**10. SPONSOR/MONITOR'S ACRONYM(S)**
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report represents the first of two reports, each of which documents a portion of a larger project. The ultimate goal of the project is to produce an end-to-end smart-trajectory model. Presented in this report is the development of a plant model that is based on the equations of motion of a maneuvering projectile. Control forces and rocket propulsion, as well as gravity, drag, and Coriolis forces, have been incorporated into the model. The purpose of the model is twofold—first, it will be used to predict future states of a projectile in order to fine-tune in-flight guidance parameters, and second, it will be the basis of an extended Kalman filter that will be used to improve upon the accuracy of the measured state of a projectile. Also included in this report is a C++ implementation of the model. The C++ implementation has been written in a matter that will both allow it to be incorporated into an extended Kalman filter and be compatible with a larger smart-trajectory model that can be run in parallel on a high-performance computing platform. A future report (the second of this pair) will contain the details of an extended Kalman filter, along with a simple sensor model and a pair of algorithms that will be useful for optimizing control parameters.

15. SUBJECT TERMS
extended Kalman filter, projectile, numeric solution, fourth-order Runge Kutta, rocket, trajectory, ballistic, C++

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Robert J. Yager
a. REPORT Unclassified	**b. ABSTRACT** Unclassified	**c. THIS PAGE** Unclassified	UU	46	**19b. TELEPHONE NUMBER** (*Include area code*) 410-278-6689

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

List of Tables

Acknowledgments

The author would like to thank Mr. Richard Pearson and Dr. Benjamin Breech of the U.S. Army Research Laboratory's Weapons and Materials Research Directorate. Mr. Pearson's many technical and editorial recommendations improved the quality of this report. Dr. Breech's advice on C++ improved the code contained in this report.

1. Introduction

This report represents the first of two reports, each of which documents a portion of a larger project. The ultimate goal of the project is to produce an end-to-end smart-trajectory model.

Presented in this report is the development of a plant (*1*) model that is based on the equations of motion of a maneuvering projectile. Control forces and rocket propulsion, as well as gravity, drag, and Coriolis forces, have been incorporated into the model. The purpose of the model is twofold—first, it will be used to predict future states of a projectile in order to fine-tune in-flight guidance parameters, and second, it will be the basis of an extended Kalman filter that will be used to improve upon the accuracy of the measured state of a projectile. Also included in this report is a C++ implementation of the model. The C++ implementation has been written in a matter that will both allow it to be incorporated into an extended Kalman filter and be compatible with a larger smart-trajectory model that can be run in parallel on a high-performance computing platform.

A future report (the second of this pair) will contain the details of an extended Kalman filter, along with a simple sensor model and a pair of algorithms that will be useful for optimizing control parameters.

2. Coordinate Systems

Three distinct coordinate systems are used for this model. A geographic coordinate system is used to locate the origin of a topocentric coordinate system with respect to a location on the surface of the earth, a topocentric coordinate system is used to track the position of a projectile as it moves through space, and a projectile-fixed coordinate system is used to orient control forces.

2.1 Geographic Coordinates

Both the atmospheric model and the Coriolis force equations that are included in this model are sensitive to variations in latitude and altitude. As a result, a geographic coordinate system is used to locate the starting location of a projectile with respect to the surface of the earth.

Due to assumed symmetry of the earth about its axis of rotation, this model is independent of longitude. Furthermore, changes in elevation are accounted for in the topocentric coordinate system, so elevation in the geographic coordinate system is fixed at sea level. Thus, although three coordinates are needed to fully specify a position in geographic coordinates, for this model, only latitude (Θ) is treated as a variable. Figure 1 shows how the geographic coordinate system is used to define the origin of the topocentric coordinate system with respect to the surface of the earth.

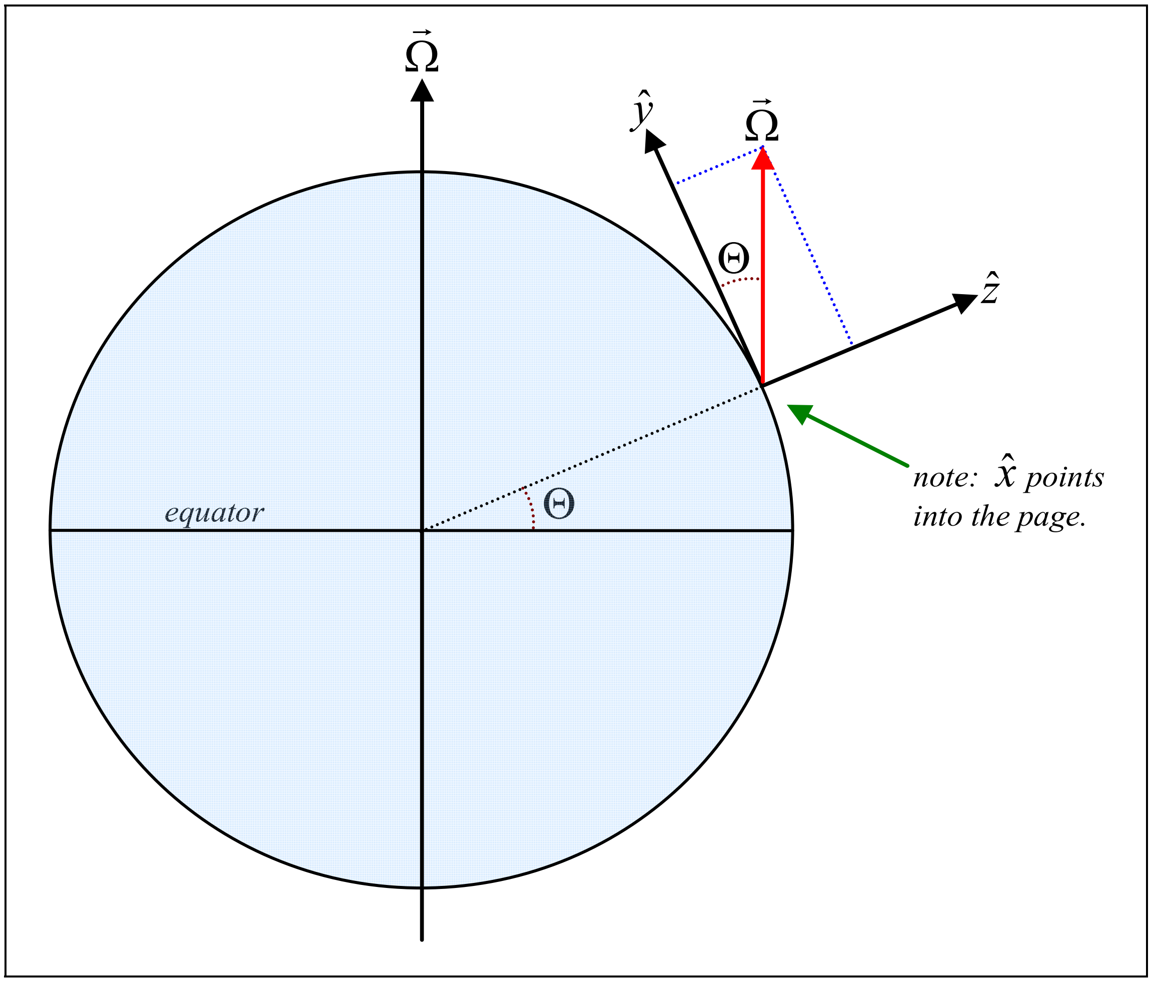

Figure 1. Relationship between geographic and topocentric coordinate systems.

2.2 Topocentric Coordinates

The equations of motion for this model are given in Cartesian, topocentric coordinates. The variables x, y, and z are used to specify a point in space in the topocentric coordinate system. Specifically, $\hat{x}$ is defined to point east, $\hat{y}$ is defined to point north, and $\hat{z}$ is defined to point in the direction of the local vertical.

Note that since the topocentric coordinate system is Cartesian, the x-y plane does not conform to the surface of the earth. A result of this is that the only locations where $\hat{x}$, $\hat{y}$, and $\hat{z}$ point *exactly* east, north, and up are along the z axis. Everywhere else, $\hat{x}$, $\hat{y}$, and $\hat{z}$ point *approximately* east, north, and up.

2.3 Projectile-Fixed Coordinates

The directions of the control forces that are included in this model are determined by a projectile's direction of travel. Thus, a projectile-fixed coordinate system is used to orient the control forces. Figure 2 shows the relationship between the projectile-fixed coordinate system and the topocentric coordinate system.

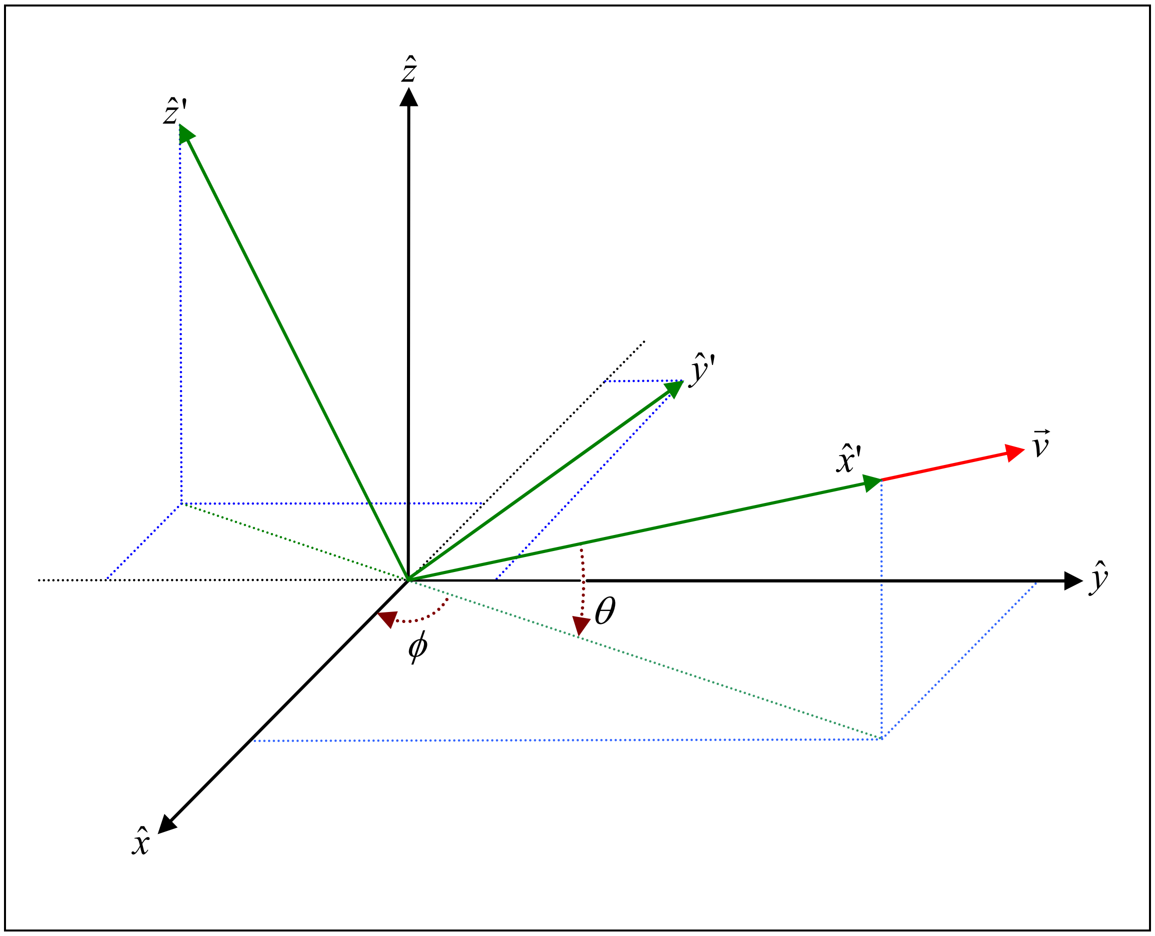

Figure 2. Relationship between topocentric and projectile-fixed coordinate systems.

The variables x', y', and z' are used to specify a point in space in the projectile-fixed coordinate system. Specifically, $\hat{x}'$ is defined to be aligned with a projectile's velocity vector ($\vec{v}$), $\hat{y}'$ is defined to point to the left of a projectile's velocity vector, and $\hat{z}'$ is defined to satisfy normal right-hand-rule conventions. In equation form, $\hat{x}'$, $\hat{y}'$, and $\hat{z}'$ are defined as follows:

$$\hat{x}' = \frac{\vec{v}}{|\vec{v}|}, \tag{1}$$

$$\hat{y}' = \frac{\hat{z} \times \vec{v}}{\left|\hat{z} \times \vec{v}\right|}, \tag{2}$$

and

$$\hat{z}' = \frac{\hat{x}' \times \hat{y}'}{\left|\hat{x}' \times \hat{y}'\right|}. \tag{3}$$

3. Coordinate Conversions

3.1 Topocentric to Geographic

Conversion from topocentric coordinates to geographic coordinates should be through an inverse azimuthal equidistant projection. Doing so will accurately preserve both distance from the origin and direction in the *x*-*y* plane. The C++ library GeographicLib (*2*) contains a function that will perform an inverse azimuthal equidistant projection.

3.2 Geographic to Topocentric

Conversion from geographic coordinates to topocentric coordinates should be through an azimuthal equidistant projection. Doing so will accurately preserve both distance from the origin and direction in the *x*-*y* plane. The C++ library GeographicLib (*2*) contains a function that will perform an azimuthal equidistant projection.

3.3 Projectile-Fixed to Topocentric

A vector given in terms of projectile-fixed coordinates can be converted to topocentric coordinates through a pair of rotations. This can be accomplished by first rotating about the *y*' axis through an angle of θ and then rotating about the new *z*' axis through an angle of ϕ. Rotation matrices can be used to perform the desired rotations. Those matrices are presented in equations 4 and 5, where $A_{y'}$ is a counterclockwise rotation about the *y*' axis by an angle of α and $A_{z''}$ is a counterclockwise rotation about the new *z*' axis by an angle of β (*3*).

$$A_{y'} = \begin{bmatrix} \cos\alpha & 0 & -\sin\alpha \\ 0 & 1 & 0 \\ \sin\alpha & 0 & \cos\alpha \end{bmatrix}, \tag{4}$$

and

$$A_{z''} = \begin{bmatrix} \cos\beta & \sin\beta & 0 \\ -\sin\beta & \cos\beta & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{5}$$

Referring to figure 2, note that θ represents a counterclockwise rotation, while ϕ represents a clockwise rotation. Thus,

$$\alpha = \theta \,, \tag{6}$$

and

$$\beta = -\phi \,. \tag{7}$$

Substituting equations 6 and 7 into equations 4 and 5 produces rotation matrices in terms of θ and ϕ.

$$A_{y'} = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix}, \tag{8}$$

and

$$A_{z''} = \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{9}$$

Note that in order to correctly apply the rotation matrices, they must be used in the proper order. This can be somewhat counterintuitive. Consider a generic vector $\vec{B}'$ that is defined in terms of the projectile-fixed coordinate system. Since $A_{y'}$ acts on $\vec{B}'$ first, it is placed immediately to the left of $\vec{B}'$. $A_{z''}$ acts on the product of $A_{y'}$ and $\vec{B}'$, so $A_{z''}$ is placed to the left of $A_{y'}$. Thus,

$$\vec{B} = A_{z''} A_{y'} \vec{B}' \,. \tag{10}$$

Equation 11 shows the product of the two rotation matrices.

$$A_{z'} A_{y'} = \begin{bmatrix} \cos\phi\cos\theta & -\sin\phi & -\cos\phi\sin\theta \\ \sin\phi\cos\theta & \cos\phi & -\sin\phi\sin\theta \\ \sin\theta & 0 & \cos\theta \end{bmatrix}. \tag{11}$$

It will be helpful to convert the right side of equation 11 into a matrix comprised of velocity components rather than sines and cosines. Trigonometric identities can be used, along with figure 2, to represent $\sin\theta$, $\cos\theta$, $\sin\phi$, and $\cos\phi$ in terms of v, v_x, v_y, and v_z. Also, the introduction of the unitless variable γ will simplify several of the derived relationships.

$$\gamma \equiv \frac{1}{\sqrt{1 - \frac{v_z^2}{v^2}}} \,, \tag{12}$$

$$\sin\theta = \frac{v_z}{v} \,, \tag{13}$$

$$\cos\theta = \frac{\sqrt{v_x^2 + v_y^2}}{v} = \frac{\sqrt{v^2 - v_z^2}}{v} = \frac{v\sqrt{1 - \frac{v_z^2}{v^2}}}{v} = \frac{1}{\gamma}, \tag{14}$$

$$\sin\phi = \frac{v_y}{\sqrt{v_x^2 + v_y^2}} = \frac{v_y}{\sqrt{v^2 - v_z^2}} = \frac{v_y}{v\sqrt{1 - \frac{v_z^2}{v^2}}} = \frac{\gamma v_y}{v}, \tag{15}$$

and

$$\cos\phi = \frac{v_x}{\sqrt{v_x^2 + v_y^2}} = \frac{\gamma v_x}{v}. \tag{16}$$

Substituting equations 13–16 into equation 11 gives a rotation matrix that is represented in terms of velocity components.

$$A_{z'}A_{y'} = \begin{bmatrix} \frac{v_x}{v} & -\frac{\gamma v_y}{v} & -\frac{\gamma v_x v_z}{v^2} \\ \frac{v_y}{v} & \frac{\gamma v_x}{v} & -\frac{\gamma v_y v_z}{v^2} \\ \frac{v_z}{v} & 0 & \frac{1}{\gamma} \end{bmatrix}. \tag{17}$$

4. Forces Acting on a Projectile

The model described in this report considers seven forces that act on a projectile. These are gravity, thrust, Coriolis, air resistance, and three control forces. Control forces are categorized as either causing a projectile to lift or turn, or as contributing to the overall air resistance of a projectile.

4.1 Gravity

In the topocentric coordinate system, gravity is assumed to always point in the $-\hat{z}$ direction. Thus,

$$\vec{F}_{grav} = m\vec{g} = -mg\hat{z}. \tag{18}$$

The value for g is determined by using the gravitational model specified in the "U.S. Standard Atmosphere" (*4*).

$$g = g_0\left(\frac{r_0}{r_0 + z}\right)^2, \tag{19}$$

where r_0 is the effective radius of the earth and g_0 is the value of g at sea level.

4.2 Thrust

It is assumed that the long axis of a projectile always points in the same direction as the projectile's velocity vector. As a result, the force due to thrust points in the same direction as the velocity vector. Thus,

$$\vec{F}_{thrust} = T\hat{v} = T\frac{\vec{v}}{v}, \tag{20}$$

where T is the magnitude of the thrust force and is typically given as a piecewise function of time.

Note that the velocity vector can be generically broken into its components in the topocentric coordinate system. Thus,

$$\vec{v} = v_x\hat{x} + v_y\hat{y} + v_z\hat{z}. \tag{21}$$

Substituting equation 21 into equation 20,

$$\vec{F}_{thrust} = \frac{Tv_x}{v}\hat{x} + \frac{Tv_y}{v}\hat{y} + \frac{Tv_z}{v}\hat{z}. \tag{22}$$

It will be helpful to define a new value whose sole purpose is to simplify equations as follows:

$$\overline{T} \equiv \frac{T}{mv}. \tag{23}$$

Substituting equation 23 into equation 22,

$$\vec{F}_{thrust} = m\overline{T}v_x\hat{x} + m\overline{T}v_y\hat{y} + m\overline{T}v_z\hat{z}. \tag{24}$$

4.3 Coriolis

The Coriolis force is a pseudo force that comes about as a result of choosing a rotating coordinate system such as the topocentric coordinate system used for this model. It is perpendicular to both the earth's angular-velocity vector ($\vec{\Omega}$) and a projectile's velocity vector ($\vec{v}$) (*5*). Specifically,

$$\vec{F}_{Coriolis} = -2m\vec{\Omega}\times\vec{v}. \tag{25}$$

Referring to figure 1, the earth's angular velocity can be described by equation 26.

$$\vec{\Omega} = \Omega\cos\Theta\,\hat{y} + \Omega\sin\Theta\,\hat{z}. \tag{26}$$

Substituting equations 21 and 26 into equation 25 and calculating the cross product,

$$\vec{F}_{Coriolis} = 2m\Omega\left(v_y \sin\Theta - v_z \cos\Theta\right)\hat{x} - 2m\Omega v_x \sin\Theta\,\hat{y} + 2m\Omega v_x \cos\Theta\,\hat{z}\ . \tag{27}$$

4.4 Air Resistance

The force due to air resistance (i.e., drag) points in a direction opposite to the direction of travel and is proportionate to the speed of the projectile squared (*6*). Specifically,

$$\vec{F}_{drag} = -\frac{1}{2}\rho A C_D v\vec{v}\ , \tag{28}$$

where ρ is the density of air at the location of a projectile, A is the cross-sectional area of a projectile, and C_D is a projectile's drag coefficient. Note that air density is calculated by the method described in the "U.S. Standard Atmosphere" (see Yager [*7*] for a summary of the process.)

Substituting equation 21 into equation 28,

$$\vec{F}_{drag} = -\frac{1}{2}\rho A C_D v v_x \hat{x} - \frac{1}{2}\rho A C_D v v_y \hat{y} - \frac{1}{2}\rho A C_D v v_z \hat{z}\ . \tag{29}$$

It will be helpful to define another new value whose sole purpose is to simplify equations:

$$\overline{C} \equiv \frac{1}{2m}\rho A C_D\ . \tag{30}$$

Substituting equation 30 into equation 29,

$$\vec{F}_{drag} = -m\overline{C} v v_x \hat{x} - m\overline{C} v v_y \hat{y} - m\overline{C} v v_z \hat{z}\ . \tag{31}$$

4.5 Turning Control Force

When control forces are used, it is assumed that some type of aerodynamic device that can cause a projectile to turn is present. Since the mechanism is assumed to be aerodynamic in nature, it is further assumed that the turning force will have a form that is similar to equation 28. Thus,

$$\vec{F}'_{turn} = \frac{1}{2}\rho\, C_{turn}\xi_{turn} v^2 \hat{y}'\ . \tag{32}$$

Two parameters are used to determine the magnitude of the turning control force. C_{turn} is projectile specific and is used to determine the maximum value of the turning control force given a particular air density and projectile speed. ξ_{turn} is situation specific and is meant to represent the proportion of the turning force being applied by the projectile, as well as the turning force direction (either left or right). Equation 33 gives the allowed values for ξ_{turn}. Note that the association between the sign of ξ_{turn} and the direction of the induced turn, left or right, is counterintuitive. A value of –1 indicates that the turning control force is at its maximum value and is causing the projectile to turn toward the $-\hat{y}'$ direction (i.e., *right*). A value of 1 indicates that the control force is at its maximum value and is causing the projectile to turn toward the $\hat{y}'$ direction (i.e., *left*). A value of 0 indicates that no turning force is being applied.

$$-1 \le \xi_{turn} \le 1 . \tag{33}$$

It will be helpful to define a new value whose sole purpose is to simplify equations. Recall that γ is defined in equation 12.

$$\overline{C}_{turn} \equiv \frac{1}{2m} \rho C_{turn} \xi_{turn} \gamma . \tag{34}$$

Substituting equation 34 into equation 32 produces a simpler equation for the turning control force.

$$\vec{F}'_{turn} = m \overline{C}_{turn} \frac{v^2}{\gamma} \hat{y}' . \tag{35}$$

Next, inserting equations 35 and 17 into equation 10 produces an equation for the turning force in terms of the topocentric coordinate system.

$$\vec{F}_{turn} = -\overline{C}_{turn} m v v_y \hat{x} + \overline{C}_{turn} m v v_x \hat{y} . \tag{36}$$

4.6 Lifting Control Force

The lifting control force is treated in much the same manner as the turning control force. The major difference is that the lifting control force points in the $\hat{z}'$ direction rather than the $\hat{y}'$ direction. Thus,

$$\vec{F}_{lift} = \frac{1}{2} \rho C_{lift} \xi_{lift} v^2 \hat{z}' , \tag{37}$$

where C_{lift} and ξ_{lift} are defined in a manner that is similar to C_{turn} and ξ_{turn}. With respect to direction, when ξ_{lift} is greater than 0, the lifting force has a component in the topocentric coordinate system that points in the $\hat{z}$ direction. When ξ_{lift} is less than zero, the lifting force has a component in the topocentric coordinates system that points in the $-\hat{z}$ direction.

Just as with the turning control force, it will be helpful to define a new equation simplifying constant. Recall that γ is defined in equation 12.

$$\overline{C}_{lift} \equiv \frac{1}{2m} \rho C_{lift} \xi_{lift} \gamma . \tag{38}$$

Substituting equation 38 into equation 37 gives the lifting control force in a simpler form.

$$\vec{F}_{lift} = m \overline{C}_{lift} \frac{v^2}{\gamma} \hat{z}' . \tag{39}$$

Next, inserting equations 39 and 17 into equation 10 gives the lifting control force in terms of the topocentric coordinate system. Note that

$$\gamma^2 = \left(\frac{1}{\sqrt{1 - \frac{v_z^2}{v^2}}} \right)^2 = \frac{v^2}{v_x^2 + v_y^2} \Rightarrow \frac{v^2}{\gamma^2} = v_x^2 + v_y^2 \tag{40}$$

and

$$\vec{F}_{lift} = -\overline{C}_{lift} m v_x v_z \hat{x} - \overline{C}_{lift} m v_y v_z \hat{y} + \overline{C}_{lift} m \left(v_x^2 + v_y^2 \right) \hat{z} \,. \tag{41}$$

4.7 Air Resistance From Control Forces

The act of applying a control force by a projectile will generally tend to increase the force due to air resistance that a projectile experiences. Equation 42 describes the additional drag caused by the application of a control force.

$$\vec{F}_{control\ \ drag} = -\left(R_{turn} F_{turn} + R_{lift} F_{lift} \right) \hat{x}' \,. \tag{42}$$

The parameters R_{turn} and R_{lift} are used to represent the proportion of the control force that contributes to an increase in drag. Thus,

$$0 \le R_{turn}, R_{lift} \le 1 \,. \tag{43}$$

From equations 1 and 21, it follows that

$$\vec{F}_{control\ \ drag} = -\left(R_{turn} F_{turn} + R_{lift} F_{lift} \right) \frac{v_x}{v} \hat{x} - \left(R_{turn} F_{turn} + R_{lift} F_{lift} \right) \frac{v_y}{v} \hat{y} - \left(R_{turn} F_{turn} + R_{lift} F_{lift} \right) \frac{v_z}{v} \hat{z}. \tag{44}$$

Just as with the turning and lifting control forces, it will be helpful to define a new equation simplifying constant.

$$\overline{C}_{control\ \ drag} \equiv \frac{1}{2m} \rho \left(R_{turn} C_{turn} \left| \xi_{turn} \right| + R_{lift} C_{lift} \left| \xi_{lift} \right| \right). \tag{45}$$

Using equations 32 and 37, equation 45 can be rewritten in terms of the magnitudes of the turning and lifting control forces.

$$\overline{C}_{control\ \ drag} = \frac{1}{m} \left(R_{turn} F_{turn} + R_{lift} F_{lift} \right) \frac{1}{v^2} \,. \tag{46}$$

Substituting equation 46 into equation 44 gives the final form of the force due to air resistance from the use of controls.

$$\vec{F}_{control\ drag} = -m\overline{C}_{control\ drag} v v_x \hat{x} + -m\overline{C}_{control\ drag} v v_y \hat{y} + -m\overline{C}_{control\ drag} v^2 \frac{v_z}{v} \hat{z}\,. \tag{47}$$

5. Equations of Motion

The equations of motion can be solved for in the standard manner by beginning with Newton's Second Law, which states that the sum of the forces acting on an object is equal to an object's mass multiplied by its acceleration.

$$\sum \vec{F} = m\vec{a}\,. \tag{48}$$

Solving equation 48 for acceleration and expanding the summation allows the spatial components of acceleration to be written in terms of the forces that were derived in section 4.

$$a_x = \frac{F_{grav,x}}{m} + \frac{F_{thrust,x}}{m} + \frac{F_{Coriolis,x}}{m} + \frac{F_{drag,x}}{m} + \frac{F_{turn,x}}{m} + \frac{F_{lift,x}}{m} + \frac{F_{control\ drag,x}}{m}\,, \tag{49}$$

$$a_y = \frac{F_{grav,y}}{m} + \frac{F_{thrust,y}}{m} + \frac{F_{Coriolis,y}}{m} + \frac{F_{drag,y}}{m} + \frac{F_{turn,y}}{m} + \frac{F_{lift,y}}{m} + \frac{F_{control\ drag,y}}{m}\,, \tag{50}$$

and

$$a_z = \frac{F_{grav,z}}{m} + \frac{F_{thrust,z}}{m} + \frac{F_{Coriolis,z}}{m} + \frac{F_{drag,z}}{m} + \frac{F_{turn,z}}{m} + \frac{F_{lift,z}}{m} + \frac{F_{control\ drag,z}}{m}\,. \tag{51}$$

Finally, substituting the previously derived force equations 18, 24, 27, 31, 35, 41, and 47 into equations 49, 50, and 51 gives the final version of the equations of motion, which are presented in table 1.

Table 1. Equations of motion.

Gravity		Thrust	Coriolis	Drag	Turn	Lift	Control-Drag
$a_x =$	0	$+\overline{T}v_x$	$+2\Omega(v_y \sin\Theta - v_z \cos\Theta)$	$-\overline{C}vv_x$	$-\overline{C}_{turn}vv_y$	$-\overline{C}_{lift}v_x v_z$	$-\overline{C}_{cdrag}vv_x$
$a_y =$	0	$+\overline{T}v_y$	$-2\Omega v_x \sin\Theta$	$-\overline{C}vv_y$	$+\overline{C}_{turn}vv_x$	$-\overline{C}_{lift}v_y v_z$	$-\overline{C}_{cdrag}vv_y$
$a_z =$	$-g$	$+\overline{T}v_z$	$+2\Omega v_x \cos\Theta$	$-\overline{C}vv_z$	$+0$	$+\overline{C}_{lift}(v_x^{\ 2} + v_y^{\ 2})$	$-\overline{C}_{Cdrag}vv_z$

The equations of motion have been solved using a fourth-order Runge-Kutta method, which is presented in code form in appendix A.

6. C++ Code

The C++ code contained in appendices A and B has been specifically designed to be embedded into larger codes. The code has been designed to be as user friendly as possible. All available calculations are performed by setting and/or modifying 21 user-definable variables and calling four functions.

6.1 User-Definable Variables

The following is a list of user-definable variables. The name of the variable is listed on the left. If the name is followed by a symbol contained in parentheses, it is the name that is used in the equations contained in this report. If applicable, the required units of each parameter are contained in square brackets.

6.1.1 Projectile Parameters

diameter_	[m] The diameter of the projectile.
mass_ (m)	[kg] The mass of the projectile.
C_turn_(C_{turn})	[m^2] The turning control coefficient.
C_lift_(C_{lift})	[m^2] The lifting control coefficient.
R_turn_(R_{turn})	[unitless] The drag scale factor for the turning control force.
R_lift_(R_{lift})	[unitless] The drag scale factor for the lifting control force.
xi_turn_(ξ_{turn})	[unitless] The portion of the turning control force used.
xi_lift_(ξ_{lift})	[unitless] The portion of the lifting control force used.
thrust_mass_table_	This table is used only when rocket propulsion is present. If used, the table contains three columns. For each row, the first-column entry represents a time (in seconds), the second-column entry represents a thrust (in Newtons), and the third-column entry represents a mass (in kilograms). Time values must be in ascending order when moving down the table, with the first value typically being 0.0 s. Values in the thrust and mass columns represent the thrust and mass values that occur for the given projectile at the time indicated by the first column. For times that fall between the values stated in the first column, thrust and mass values are determined by performing a linear interpolation.

drag_table_	This table provides a piecewise function that can be used to determine the zero-yaw drag coefficient. The model is designed to accept standard zero-yaw drag coefficient tables such as those found in U.S. Department of Defense firing table manuals. (See FCI 155-AO-A for an example [*8*]). In these tables, the drag coefficients are given as polynomials in term of Mach number. The last column of the table contains Mach numbers and is used to determine which row (i.e., which function) should be used to calculate the zero-yaw drag coefficient, where the Mach number given represents the maximum Mach number allowable for that row. The minimum Mach number allowable for a given row is given by the Mach number contained in the previous row, or zero if no previous row exists. All other columns represent the unitless coefficients of the polynomial describing the zero-yaw drag coefficient. Once the proper row is determined, the equation $C_D = \sum_{i=0}^{n-1} a_i M^i$ is used, where M is the projectile's current Mach number, n is the number of columns in the drag_table_ variable, and a_i is the value of the polynomial coefficient given in the *i*th column of the table.

6.1.2 Environmental Parameters and Initial Conditions

t0_	[s] The initial time offset.
x0_	[m] The initial spatial offset in the $\hat{x}$ direction.
y0_	[m] The initial spatial offset in the $\hat{y}$ direction.
z0_	[m] The initial spatial offset in the $\hat{z}$ direction.
v0_	[m/s] The initial speed of a projectile.
elevation0_	[radians] The angle between the initial velocity vector and the horizontal plane. If the initial velocity vector has a positive z component, then this angle will be positive.
azimuth0_	[radians] The angle between the projection of the initial-velocity vector onto the x-y plane and the x axis, measured counterclockwise.
lat_ (Θ)	[radians] The latitude of the origin of the topocentric coordinate system.
T0_	[K] The temperature at the origin of the topocentric coordinate system.
P0_	[Pa] The pressure at the origin of the topocentric coordinate system.

6.1.3 Other

delta_t_calc_	[s] The time step that is used in calculating the end state and the state at apogee.

6.2 State Vectors

State vectors are class members of the defined C++ container class *yVector*, which is presented in appendix B. The class is designed to function similar to the built-in C++ container class *vector <double>*. All state vectors contain 10 elements and are used to represent the state of the projectile in the topocentric coordinate system. Table 2 provides a cross-referenced list between a state vector's element numbers and the values contained in a state vector.

Table 2. Description of the elements contained in a state vector.

Element No.	Description	Units
0	time	s
1	x	m
2	y	m
3	z	m
4	vx	m/s
5	vy	m/s
6	vz	m/s
7	ax	m/s^2
8	ay	m/s^2
9	az	m/s^2

6.3 Functions

Four functions are available for calculating various state vectors. Three accept state vectors as arguments, all four return state vectors. (Refer to section 6.2 for a description of state vectors.)

6.3.1 Initial()

This function calculates the initial state of a projectile based solely on user-defined parameters.

6.3.2 Next()

Given the current state of a projectile, this function calculates the state of the projectile at some given time after the current time.

6.3.3 Apogee()

Given the current state of a projectile, this function calculates the state of the projectile at apogee. Note that the user-defined variable delta_t_calc_ is used to determine the time step between iterations.

6.3.4 End()

Given the current state of a projectile, this function calculates the state of the projectile at its end state. The end state is defined to be the state of the projectile when it returns to ground level (i.e., when $z = 0$ and $v_z < 0$). Note that the user-defined variable delta_t_calc_ is used to determine the time step between iterations.

6.4 Using the Code

Section A.1 of appendix A presents a simple example of how the projectile model can be included in custom-written C++ code. Projectile parameters, environmental parameters, and initial conditions are easily defined by accessing class-member variables. Four simple functions provide flexibility for the model. All four functions return a state vector that describes the position, velocity, and acceleration of a projectile at a given time.

7. References

1. Ogata, K. *Modern Control Engineering*, 4th ed.; Prentice Hall: Upper Saddle River, NJ, 2002.

2. Karney, C. GeographicLib. http://sourceforge.net/projects/geographiclib/ (accessed 8 February 2011).

3. Arfken, G. B.; Weber, H. J. *Mathematical Methods for Physics*, 5th ed.; Academic Press: San Diego, CA, 2001.

4. National Aeronautics and Space Administration. *U.S. Standard Atmosphere*; NASA-TM-X-74335; U.S. Government Printing Office: Washington, DC, 1976.

5. Goldstein, H.; Poole, C.; Safko, J. *Classical Mechanics*, 3rd ed.; Addison Wesley: San Francisco, CA, 2002.

6. McCoy, R. L. *Modern Exterior Ballistics*; Schiffer Publishing Ltd.: Atglen, PA, 1999.

7. Yager, R. J. *A Two-Dimensional, Numeric Technique for Approximating the Trajectory of a Rocket/Projectile Using C++*; ARL-TR-4608; U.S. Army Research Laboratory: Aberdeen Proving Ground, MD, 2008.

8. U.S. Army Armament Research, Development, and Engineering Center. Howitzer Medium, Self-Propelled, 155mm, M109A1, M109A1B, M109A2, M109A3, M109A3B, and M109A4; and Howitzer, Medium, Towed, 155mm, M198; Howitzer, Medium, Self-Propelled, 155mm, M109A5 and M109A6; Howitzer, Medium, Towed, 155mm, M777, M777A1 and M777A2 Firing Projectiles, HERA, M549 and M549A1; FCI 155-AO-A; Aberdeen Proving Ground, MD, 2007.

Appendix A. Trajectory Code

This appendix appears in its original form, without editorial change.

A.1 y_traj_main.cc

```
#include "y_traj_class.h"
/******************************************************************************/
int main(){
  yTraj T;
  //---------------ASSIGN VALUES TO USER-DEFINABLE VARIABLES--------------------
  //PROJECTILE PARAMETERS
  T.diameter_=1.00;//........................... ....diameter of projectile (m)
  T.mass_=30.000;//...................................mass of projectile (kg)
  T.C_turn_=0.01;//...........................turning-control coefficient (m^2)
  T.C_lift_=0.01;//...........................lifting-control coefficient (m^2)
  T.R_turn_=0.5;//.....................turning-control-to-drag ratio (unitless)
  T.R_lift_=0.5;//.....................lifting-control-to-drag ratio (unitless)
  T.xi_turn_=0.0;//................................turning proportion(unitless)
  T.xi_lift_=0.0;//................................lifting proportion(unitless)
  T.thrust_mass_table_="";//..........................use for rocket propulsion
  T.drag_table_=//...................................obtain from firing tables
    " 0.10000000e+00 ,  0.00000000e+00 , 1.00000000e+00 ;"
    " 0.29000000e-01 ,  0.11400000e+00 , 5.00000000e+00";
  //ENVIRONMENTAL PARAMETERS AND INITIAL CONDITIONS
  T.t0_=0.0;//.................................................time offset (s)
  T.x0_=0.0;//....................initial spatial offset in the x-direction (m)
  T.y0_=0.0;//....................initial spatial offset in the y-direction (m)
  T.z0_=0.0;//....................initial spatial offset in the z-direction (m)
  T.v0_=700.0;//.............................................initial speed (m/s)
  T.elevation0_=20.0*M_PI/3200.0;//.........initial angle of elevation (radians)
  T.azimuth0_=1600.0*M_PI/3200.0;//.............initial azimuth angle (radians)
  T.lat_=45.5*M_PI/180.0;//........................latitude at origin (radians)
  T.T0_=288.15;//............................temperature at sea level (Kelvins)
  T.P0_=101325.0;//..............................pressure at sea level (Pascals)
  //OTHER PARAMETERS
  T.delta_t_calc_=0.1;//...time step used to calculate apogee and end states (s)
  /*----------------------------------------------------------------------------
                            STATE-ELEMENT KEY
                   (ALL UNITS ARE IN METERS AND SECONDS)

                 ELEMENT: 0, 1, 2, 3,  4,  5,  6,  7,  8,  9
                VARIABLE: t, x, y, z, vx, vy, vz, ax, ay, az
  ----------------------------------------------------------------------------*/
  //--------------------------CALCULATE INITIAL STATE--------------------------
  yVector initial=T.Initial();
  //-------------------------CALCULATE STATE AT APOGEE-------------------------
  yVector apogee=T.Apogee(initial);
  //---------------------------CALCULATE END STATE-----------------------------
  yVector end=T.End(T.Initial());
  //------------------------CALCULATE ENTIRE TRAJECTORY------------------------
  yMatrix full=T.Initial();
  for(int i=0;full[i][3]>0||full[i][6]>0;++i)//...........Do while z>0 or vz>0.
    full+=T.Next(0.1,full[i]);
  return 0;
}/****************************************************************************/
```

A.2 y_traj_class.h

```
   /*************************************************************************
 ****************************************************************************
***                                                                       ***
**                             TRAJECTORY CLASS                            **
**                         version 1.30 (04-07-2011)                       **
**                                -Rob Yager                               **
***                                                                       ***
 ****************************************************************************
    *************************************************************************
This class uses a 3DoF model to calculate trajectories.  Control forces and
rocket propulsion are included.
****************************************************************************/
#ifndef Y_TRAJ_CLASS_H_
#define Y_TRAJ_CLASS_H_
/***************************************************************************/
#include "y_atmosphere_class.h"
#include "..\general\y_matrix_class.h"
#include "..\general\y_math_namespace.h"
/***************************************************************************/
class yTraj:public yAtmosphere{
public://----------------------PUBLIC PARAMETERS------------------------------
  //PROJECTILE PARAMETERS
  double diameter_;//.......................diameter of the projectile (meters)
  double mass_;//................................mass of projectile (kilograms)
  double C_turn_;//.....control coefficient for yp-direction control force (m^2)
  double C_lift_;//.....control coefficient for zp-direction control force (m^2)
  double R_turn_;//...................yp-direction drag scale factor (unitless)
  double R_lift_;//...................zp-direction drag scale factor (unitless)
  double xi_turn_;//portion of control used (-1 to 1) for y-direction (unitless)
  double xi_lift_;//portion of control used (-1 to 1) for z-direction (unitless)
  yMatrix thrust_mass_table_;//.....contains thrust and mass piecewise functions
  yMatrix drag_table_;//.....drag table specific to the projectile being modeled
  //ENVIRONMENTAL PARAMETERS AND INITIAL CONDITIONS
  double t0_;//.........................................time offset (seconds)
  double x0_;//........................................initial x value (meters)
  double y0_;//........................................initial y value (meters)
  double z0_;//........................................initial z value (meters)
  double v0_;//...................................initial speed (meters/second)
  double elevation0_;//............................angle of elevation (radians)
  double azimuth0_;//...initial azimuth (radians, measured clockwise from north)
  double lat_;//..................................................latitude (radians)
  //OTHER PARAMETERS
  double delta_t_calc_;//..................time step for calculations (seconds)
  //--------------------------PUBLIC FUNCTIONS-------------------------------
  yTraj();//................................................default constructor
  yVector Initial() const;//........................calculates the state at t=0
  yVector Next(double delta_t,const yVector &state) const;//..state at next time
  yVector Apogee(const yVector &initial) const;//........returns state at apogee
  yVector End(const yVector &initial) const;//........state at end of trajectory
  double f(int i,const yVector &x,double deltat) const;//needed for Kalman filt.
private://------------------------PRIVATE------------------------------------
  static const int n=10;//....................number of elements in state vector
  double r(int i,double delta_t,const yVector &state) const;//...pos. components
  yVector v(double delta_t,const yVector &x) const;//...........velocity vector
  yVector a(double delta_t,const yVector &x) const;//........acceleration vector
};/*************************************************************************/
#endif/*********************************************************************/
```

A.3 y_traj_class.cc

```
//***************Rob Yager, version 1.30 (04-07-2011)***************************
#include "y_traj_class.h"
/******************************************************************************/
yVector yTraj::Initial() const{
  //------------------MAKE SURE elevation0_ ISN'T +-PI/2------------------------
  double el=elevation0_;
  if(elevation0_==M_PI/2.0||elevation0_==-M_PI/2.0) el+=10e-8;
  //----------------SETUP (THEN RETURN) THE INITIAL STATE-----------------------
  yVector x(10);
  x[0]=t0_;//...................................................initial time (s)
  x[1]=x0_;//............................................initial east offset (m)
  x[2]=y0_;//...........................................initial north offset (m)
  x[3]=z0_;//..............................................initial up offset (m)
  x[4]=v0_*cos(el)*sin(azimuth0_);//..................................vx0 (m/s)
  x[5]=v0_*cos(el)*cos(azimuth0_);//..................................vy0 (m/s)
  x[6]=v0_*sin(el);//.................................................vz0 (m/s)
  return x;//.........return the state at the start condition (note ax,ay,az=0)
}/******************************************************************************/
yVector yTraj::Next(double delta_t,const yVector &state) const{
  //--------CALCULATE ACCELERATION AND VELOCITY VECTORS AT NEW TIME-------------
  yVector v=(*this).v(delta_t,state);//..........................velocity vector
  yVector a=(*this).a(delta_t,state);//......................acceleration vector
  //--------------SETUP (THEN RETURN) THE NEW STATE VECTOR----------------------
  yVector x(10);
  x[0]=state[0]+delta_t;
  for(int i=0;i<3;++i) x[i+7]=a[i],x[i+4]=v[i],x[i+1]=r(i,delta_t,state);
  return x;
}/******************************************************************************/
yVector yTraj::Apogee(const yVector &initial) const{
  if(initial[6]<=0) return initial;//can't predict apogee if it's already passed
  //----------------FIND THE STATES BEFORE AND AFTER APOGEE---------------------
  yMatrix x(2,10);
  x[0]=initial;
  while(x[0][6]>0) x[1]=x[0],x[0]=Next(delta_t_calc_,x[0]);
  //--------------CALCULATE (AND RETURN) THE STATE AT APOGEE--------------------
  yVector apogee(10);
  for(int i=0;i<10;++i) apogee[i]=yMath::LinInterp(x.Col(6),x.Col(i),0.0);
  return apogee;//..........................................the state at apogee
}/******************************************************************************/
yVector yTraj::End(const yVector &initial) const{
  if(initial[3]<=0&&initial[6]<=0) return initial;//.can't predict end if passed
  //-------------FIND THE STATES BEFORE AND AFTER THE END STATE-----------------
  yMatrix x(2,10);
  x[0]=initial;
  while(x[0][3]>0||x[0][6]>0) x[1]=x[0],x[0]=Next(delta_t_calc_,x[0]);
  //-----------------CALCULATE (AND RETURN) THE END STATE-----------------------
  yVector end(10);
  for(int i=0;i<10;++i) end[i]=yMath::LinInterp(x.Col(3),x.Col(i),0.0);
  return end;
}/******************************************************************************/
double yTraj::f(int i,const yVector &x,double deltat) const{
  if(i==0) return x[0]+deltat;//..................time component of state vector
  if(i<4) return r(i-1,deltat,x);//.........position component of state vector
  if(i<7) return v(deltat,x)[i-4];//.........velocity component of state vector
  return a(deltat,x)[i-7];//..............acceleration component of state vector
}/******************************************************************************/
double yTraj::r(int i,double delta_t,const yVector &state) const {
  return .5*state[7+i]*delta_t*delta_t+state[4+i]*delta_t+state[1+i];
}/******************************************************************************/
yVector yTraj::v(double delta_t,const yVector &state) const{
  //----------USE A FOURTH ORDER RUNGE-KUTTA SOLVER TO FIND VELOCITY------------
  yVector out=state;
  yVector k1=a(0,state);
  for(int j=0;j<3;++j) out[j+4]=state[j+4]+.5*delta_t*k1[j];
  yVector k2=a(.5*delta_t,out);
  for(int j=0;j<3;++j) out[j+4]=state[j+4]+.5*delta_t*k2[j];
  yVector k3=a(.5*delta_t,out);
  for(int j=0;j<3;++j) out[j+4]=state[j+4]+delta_t*k3[j];
  yVector k4=a(delta_t,out);
  yVector v(3);
  for(int j=0;j<3;++j)
    v[j]=state[j+4]+(delta_t/6.0)*(k1[j]+k2[j]*2+k3[j]*2+k4[j]);
  return v;
}/*************************************************************************/
yVector yTraj::a(double delta_t,const yVector &x) const{
  //--------------IF delta_t=0 RETURN THE CURRENT ACCELERATION------------------
  yVector a(3);
  if(delta_t==0){
    for(int i=0;i<3;++i) a[i]=x[i+7];
    return a;
  }
  //--------------PRE-CALCULATE VALUES NEEDED FOR ACCELERATION------------------
  const double omega=.000072921159;//.........rotational speed of the earth (/s)
  double A=pow(diameter_/2,2)*M_PI;//.................cross-sectional area (m^2)
  double m=mass_;//....................................................mass (kg)
  if(thrust_mass_table_.Size(0)!=0)
    m=yMath::LinInterp(thrust_mass_table_.Col(0),
                       thrust_mass_table_.Col(2),x[0]);
  double T=0;//.................................................thrust (Newtons)
  if(thrust_mass_table_.Size(0)!=0)
    T=yMath::LinInterp(thrust_mass_table_.Col(0),
                      thrust_mass_table_.Col(1),x[0]);
  double v=sqrt(x[4]*x[4]+x[5]*x[5]+x[6]*x[6]);//....................speed (m/s)
  double gamma=pow(1-x[6]*x[6]/(v*v),-0.5);//....simplifying constant (unitless)
  if(v==0.0||v==x[6]) gamma=1;
  double M=Mach(v,x[3]);//...............................mach number (unitless)
  double C=0;//.................................................drag coefficient
  if(M>drag_table_[drag_table_.Size(0)-1][drag_table_.Size(1)-1])//            |
    Error("Maximum Mach level exceded.");                          //            |
  for(int j=0;j<drag_table_.Size(0);++j){                          //            |
    if(M<=drag_table_[j][drag_table_.Size(1)-1]){                  //            |
      for(int k=0;k<drag_table_.Size(1)-1;++k)                     //            |
         C+=drag_table_[j][k]*pow(M,k);                            //            |
      j=drag_table_.Size(0);                                       //            |
    }                                                              //            V
  }//---------------------------------------------------------------------------
  double rho=Density(x[3]);//..............................air density (kg/m^3)
  double Tbar=T/(m*v);//...........................equation simplifying constant
  double Cbar=1/(2.0*m)*rho*A*C;//.................equation simplifying constant
  double Cybar=.5*rho*C_turn_*xi_turn_*gamma/m;//..equation simplifying constant
  double Czbar=.5*rho*C_lift_*xi_lift_*gamma/m;//..equation simplifying constant
  double Cxbar=.5*rho*(R_turn_*C_turn_//...........equation simplifying constant
    *abs(xi_turn_)+R_lift_*C_lift_*abs(xi_lift_))/m;
  //---------------CALCULATE AND RETURN THE ACCELERATION VECTOR-----------------
  a[0]=0
       +Tbar*x[4]                                  //..................Thrust Term
       +2*omega*(x[5]*sin(lat_)-x[6]*cos(lat_))    //.....................Coriolis
       -Cbar*v*(x[4])                              //....................Drag Term
       -Cybar*v*(x[5])                             //..............y control force
       -Czbar*(x[4])*x[6]                          //..............z control force
       -Cxbar*v*(x[4]);                            //..............x control force
  a[1]=0
       +Tbar*x[5]                                  //..................Thrust Term
       -2*omega*x[4]*sin(lat_)                     //.....................Coriolis
       -Cbar*v*(x[5])                          //....................Drag Term
       +Cybar*v*(x[4])                         //..............y control force
       -Czbar*(x[5])*x[6]                      //..............z control force
       -Cxbar*v*(x[5]);                        //..............x control force
  a[2]=-g(x[3])
       +Tbar*x[6]                              //..................Thrust Term
       +2*omega*x[4]*cos(lat_)                 //.....................Coriolis
       -Cbar*v*x[6]                            //....................Drag Term
       +0                                      //..............y control force
       +Czbar*(pow(x[4],2)+pow(x[5],2))        //..............z control force
       -Cxbar*v*x[6];                          //..............x control force
  return a;
}/******************************************************************************/
```

A.4 y_atmosphere_class.h

```
   /***************************************************************************
 ******************************************************************************
***                                                                          ***
**                            ATMOSPHERE CLASS                                **
**                        version 1.10 (01-23-2011)                           **
**                               -Rob Yager                                   **
***                                                                          ***
 ******************************************************************************
    ***************************************************************************
The functions contained in this class can be used to calculate temperature,
pressure, density, and speed of sound for specified altitudes.  This code is an
implementation of the model developed in "U.S. Standard Atmosphere, 1976."

Height Limits:
  Do not use the atmospheric functions contained in this class for altitudes
  above 86,000 m.  Also, the atmospheric model used to create the functions in
  this class is not guaranteed to be accurate below 5000 meters below sea level.

Temperature Function:
  The temperature that is calculated is actually the molecular-scale temperature
  (rather than the kinetic temperature).  However, up to about 80,000 m the two
  are identical.  From 80,000 m to 86,000 m the difference between the two is
  trivial.  The largest difference occurs at 86,000 m and is about 0.008%.  See
  page 9 of "U.S. Standard Atmosphere, 1976" for more details.
*******************************************************************************/
#ifndef Y_ATMOSPHERE_CLASS_H_
#define Y_ATMOSPHERE_CLASS_H_
/******************************************************************************/
#include <math.h>
#include "..\General\y_error_namespace.h"
using yError::Error;
/******************************************************************************/
class yAtmosphere{
public://----------------------PUBLIC PARAMETERS-------------------------------
  double T0_;//..............................temperature at sea level (Kelvins)
  double P0_;//...................................pressure at sea level (Pascals)
  //--------------------------PUBLIC FUNCTIONS--------------------------------
  yAtmosphere();//...........................................default constructor
  double T(double z) const;//......temperature (K) as a function of altitude (m)
  double P(double z) const;//...pressure (Pascals) as a function of altitude (m)
  double Density(double z) const;//...density (kg/m^3) as a function of alt. (m)
  double SpeedofSound(double z) const;//....speed of sound (m/s) of altitude (m)
  double Mach(double v,double z) const;//mach(no unit) of alt.(m) and speed(m/s)
  double g(double z) const;//......gravity (m/s^2) as a function of altitude (m)
private://------------------------PRIVATE-------------------------------------
  static const double table4_[8][2];
};/*****************************************************************************/
#endif/************************************************************************/
```

A.5 y_atmosphere_class.cc

```
#include "y_atmosphere_class.h"
/*****************************************************************************/
#define RE 6356766.0//.....................The radius of the earth (m), [page 8]
#define G0 9.80665//......................sea-level value of g (m/s^2), [page 8]
#define M0 28.9644//...................mean molecular weight (kg/kmol), [page 9]
#define RSTAR 8314.32//..........universal gas constant (N*m/(kmol*K)), [page 3]
#define GAMMA 1.4//................heat capacity ratio (dimensionless), [page 4]
/*****************************************************************************/
const double yAtmosphere::table4_[8][2]={//.The first column is the geopotential
  00000.00  , -0.0065 ,                 //    height (m'), the second column is
  11000.00  ,  0.0000 ,                 //      the molecular-scale temperature
  20000.00  ,  0.0010 ,                 //           gradient (K/m'), [page 3].
  32000.00  ,  0.0028 ,
  47000.00  ,  0.0000 ,
  51000.00  , -0.0028 ,
  71000.00  , -0.0020 ,
  84852.05  ,  0.0000
};/***************************************************************************/
yAtmosphere::yAtmosphere(){
}/***************************************************************************/
double yAtmosphere::T(double z) const{
  if(z>86000) Error("86,000 m altitude limit has been exceeded.");
  double T=T0_,Hb=0.0,H=1.0/(1.0/z+1.0/RE);//.............note H(z)=1/(1/z+1/RE)
  bool exit=false;
  for(int b=0;!exit;++b){
    if(H>table4_[b+1][0]) Hb=table4_[b+1][0];
    else Hb=H,exit=true;
    T+=table4_[b][1]*(Hb-table4_[b][0]);//................[page 10, equation 23]
  }
  return T;//...............................................Temperature (Kelvins)
}/***************************************************************************/
double yAtmosphere::P(double z) const{
  if(z>86000) Error("86,000 m altitude limit has been exceeded.");
  double P=P0_,Hb=0.0,H=1.0/(1.0/z+1.0/RE);//.............note H(z)=1/(1/z+1/RE)
  bool exit=false;
  for(int b=0;!exit;++b){
    double Tb=T(1.0/(1.0/table4_[b][0]-1.0/RE));//........note z(H)=1/(1/H-1/RE)
    if(H>table4_[b+1][0]) Hb=table4_[b+1][0];
    else Hb=H,exit=true;
    if(table4_[b][1]!=0){
      P*=pow(Tb/(Tb+table4_[b][1]*(Hb-table4_[b][0])),//.[page 12, equation 33a]
        G0*M0/(RSTAR*table4_[b][1]));
    }
    else{
      P*=exp(-G0*M0*(Hb-table4_[b][0])/(RSTAR*Tb));//....[page 12, equation 33b]
    }
  }
  return P;//..................................................Pressure (Pascals)
}/***************************************************************************/
double yAtmosphere::Density(double z) const{
  return P(z)*M0/(RSTAR*T(z));//........Density (kg/m^3), [Page 15, equation 42]
}/***************************************************************************/
double yAtmosphere::SpeedofSound(double z) const{
  return sqrt(GAMMA*RSTAR*T(z)/M0);//..............(m/s), [Page 18, equation 50]
}/***************************************************************************/
double yAtmosphere::Mach(double v,double z) const{
  return v/SpeedofSound(z);//.....................................(dimensionless)
}/***************************************************************************/
double yAtmosphere::g(double z) const{
  return G0*pow(RE/(RE+z),2);//...accel. due to g (m/s^2), [page 8, equation 17]
}/***************************************************************************/
```

```
#undef RE
#undef G0
#undef M0
#undef RSTAR
#undef GAMMA
```

Appendix B. Utilities Code

This appendix appears in its original form, without editorial change.

B.1 y_error_namespace.h

```
  /*************************************************************************
 *****************************************************************************
***                                                                         ***
**                              ERROR FUNCTIONS                              **
**                          version 1.00 (01-23-2011)                        **
**                                 -Rob Yager                                **
***                                                                         ***
 *****************************************************************************
    ************************************************************************/
#ifndef Y_ERROR_NAMESPACE_H_
#define Y_ERROR_NAMESPACE_H_
/*******************************************************************************/
#include <string>
#include <iostream>
using std::string;
/*******************************************************************************/
namespace yError{
  void Warning(string message);//.........displays a variety of warning messages
  void Error(string message);//.............displays a variety of error messages
};/*****************************************************************************/
#endif/************************************************************************/
```

B.2 y_error_namespace.cc

```
#define _CRT_SECURE_NO_WARNINGS//..................disables deprecation warnings
#include "y_error_namespace.h"
/*******************************************************************************/
void yError::Warning(std::string message){
  printf("\n\nWarning:\n%s\n\n",message.c_str());
}/******************************************************************************/
void yError::Error(std::string message){
  printf("\n\nError:\n%s\n\n",message.c_str());
  printf("Press enter to continue . . .");
  std::cin.ignore(1);
  exit(1);
}/******************************************************************************/
```

B.3 y_math_namespace.h

```
  /*************************************************************************
 *****************************************************************************
***                                                                         ***
**                               MATH FUNCTIONS                              **
**                          version 1.10 (01-23-2011)                        **
**                                 -Rob Yager                                **
***                                                                         ***
 *****************************************************************************
    ************************************************************************/
#ifndef Y_MATH_NAMESPACE_H_
#define Y_MATH_NAMESPACE_H_
/*******************************************************************************/
#ifndef M_PI
#define M_PI 3.141592653589793
#endif
/*******************************************************************************/
#include "y_vector_class.h"
/*******************************************************************************/
namespace yMath{
  //------------------------Random Number Generators----------------------------
  int RandInt(int lower,int upper);//...........returns a uniformly distributed
                                          //                       random integer
  double RandDouble(double lower,  //...........returns a uniformly distributed
                         double upper); //                          random double
  double NormRand(double mu,       //............returns a normally distributed
                       double sigma);   //                          random double
  //-----------------------------------Other-------------------------------------
  double ToRadians(double angle,string units);//.......units "degrees" or "mils"
  double FromRadians(double angle,string units);//.......... "degrees" or "mils"
  double LinInterp(yVector x,yVector function,double x_interpolate);
  int LinInterpInputChecker(yVector x,yVector f);
  double PercentError(double measured,//...............calculates percent error
                           double actual);//             of the measured value
  double PercentDifference(double a,//.......................returns the percent
                                double b); //       difference between a and b
  double Round(double a,int decimals);//........round-half-away-from-zero method
  double Round2(double a,int sigs);//......uses.round-half-away-from-zero method
  double Hypot(double x,double y);//.....................2D hypotenuse function
  double Hypot(double x,double y,double z);//.............3D hypotenuse function
};/***************************************************************************/
#endif/*************************************************************************/
```

B.4 y_math_namespace.cc

```
#include "y_math_namespace.h"
/******************************************************************************/
int yMath::RandInt(int lower,int upper){
  return (int)floor((upper-lower)*(rand()+1)/(RAND_MAX+1.0))+lower;
}/*****************************************************************************/
double yMath::RandDouble(double lower,double upper){
  return lower+(upper-lower)*rand()/RAND_MAX;
}/*****************************************************************************/
double yMath::NormRand(double mu, double sigma){
  double u=(rand()+1)/(RAND_MAX+1.0);//........a random number in interval (0,1]
  double v=(rand()+1)/(RAND_MAX+1.0);//........a random number in interval (0,1]
  return sqrt(-2*log(u))*cos(2*M_PI*v)*sigma+mu;//..........Box-Muller transform
}/*****************************************************************************/
double yMath::ToRadians(double angle,string units){
  if(units=="degrees")  return angle*M_PI/180.;
  if(units=="mils")  return angle*M_PI/3200.;
  Error("Unable to convert to radians, incorrect units specified.");
}/*****************************************************************************/
double yMath::FromRadians(double angle,string units){
  if(units=="degrees") return angle*180./M_PI;
  if(units=="mils") return angle*3200./M_PI;
  Error("Unable to convert from radians, incorrect units specified.");
}/*****************************************************************************/
double yMath::LinInterp(yVector x,yVector f,double x_interpolate){ /*
Note that all elements of the x vector must be unique and in ascending order.
Use the yLinInterpInputChecker() function to make sure that the x and f inputs
are sized consistently and that x is ordered correctly.*/
  if(x_interpolate<x[0]||x_interpolate>x[x.Size()-1])
    Error("Interpolation value is out of range.");
  //Find the x-values that bracket x_interpolate.  (Note that starting at one
  //takes care of the x_interpolate = x[0] case.)
  int i;
  for(i=1;x[i]<x_interpolate;++i);//........... x[i-1] <= x_interpolate <= x[i]
  //Perform interpolation.
  return f[i-1] + (x_interpolate - x[i-1])*(f[i]-f[i-1])/(x[i]-x[i-1]);
}/*****************************************************************************/
int yMath::LinInterpInputChecker(yVector x,yVector f){   /*
Use this function to check the x and f inputs of the yLinInterp() function.
Note that the yLinInterpInputChecker() function only needs to be ran initially
and when the x or f inputs are changed.
RETURN CODES:
  0 - no errors
  1 - size mismatch between x-vector and f-vector
  2 - x-vector failed unique/ascending-order test
Note that if multiple errors are present, only one error code will be returned.
------------------------------------------------------------------------------*/
  //Check to see if the x-vector and the f-vector are sized correctly.
  if(x.Size()!=f.Size()) Error("Interpolator size mismatch");
  //Check to see if all x-values are unique and in ascending order.
  for(int i=1;i<(int)x.Size();++i) if(x[i]<=x[i-1])
    Error("Values out of order");
  return 0;
}/*****************************************************************************/
double yMath::PercentError(double actual, double measured){
  return 100*(actual-measured)/actual;
}/*****************************************************************************/
double yMath::PercentDifference(double a,double b){
  return 100.0*abs(a-b)/((a+b)/2.0);
}/*****************************************************************************/
double yMath::Round(double a,int decimals){
  double sign=1.0;
  if(a<0) sign=-1;//......................................remember the sign of a
  if(a==0) return 0;//............................handle special case where a=0
  double base=sign*a*pow(10.0,decimals);
  if(base-(int)base>=.5) base+=1;//........................perform rounding step
  return sign*(int)base*pow(10.0,-decimals);
}/*****************************************************************************/
double yMath::Round2(double a,int sigs){
  double sign=1.0;
  if(a<0) sign=-1;//......................................remember the sign of a
  if(a==0) return 0;//............................handle special case where a=0
  if(sign*a>0&&sign*a<1) sigs++;//.......handle case where a is between -1 and 1
  int power=(int)(log(sign*a)/log(10.0));
  double base=(sign*a*pow(10.0,sigs-1-power));
  if(base-(int)base>=.5) base+=1;//........................perform rounding step
  return sign*(int)base*pow(10.0,power-sigs+1);
}/*****************************************************************************/
double yMath::Hypot(double x,double y){
  return sqrt(x*x+y*y);
}/*****************************************************************************/
double yMath::Hypot(double x,double y,double z){
  return sqrt(x*x+y*y+z*z);
}/*****************************************************************************/
```

B.5 y_vector_class.h

```
   /************************************************************************
 ******************************************************************************
***                                                                          ***
**                           VECTOR CONTAINER CLASS                           **
**                         version 1.10 (01-23-2011)                          **
**                                -Rob Yager                                  **
***                                                                          ***
 ******************************************************************************
    ************************************************************************/
#ifndef Y_VECTOR_CLASS_H_
#define Y_VECTOR_CLASS_H_
/*****************************************************************************/
#include <vector>
#include <math.h>
```

```
#include "y_error_namespace.h"
using std::vector;
using std::string;
using yError::Error;
/******************************************************************************/
class yVector{
public://---------------------PUBLIC FUNCTIONS--------------------------------
  yVector();//...................default constructor (size is initially unknown)
  yVector(int size);//....................constructor (size is initially known)
  yVector(const string &vec);//...constructor (entire vector is initially known)
  int Size() const;//.................returns the number of elements in a vector
  void Resize(int size);//.........................changes the size of a vector
  void Show() const;//.........................displays the contents of a vector
  void Sort();//...............puts the contents of a vector in increasing order
  double yMin() const;//...returns smallest (or most negative) value in a vector
  double yMax() const;//...................returns the largest value in a vector
  double Avg() const; //.......returns the average of all the values in a vector
  yVector Abs() const;//....................returns a vector of absolute values
  //-------------------------OPERATOR OVERLOADS------------------------------
  double &operator[](int i);//...............returns the ith element of a vector
  double operator[](int i) const;//..........returns the ith element of a vector
  yVector operator+=(double new_element);//...adds an element to end of a vector
  yVector operator=(const string &elements);//................populates a vector
  yVector operator*(double right) const;//.................scalar multiplication
  yVector operator+(const yVector &right) const;//...............vector addition
  yVector operator-(const yVector &right) const;//............vector subtraction
private://------------------------PRIVATE----------------------------------
  int size_;//................................the number of elements in a vector
  vector <double> store_;//.....................storage for all vector elements
};/****************************************************************************/
#endif/************************************************************************/
```

B.6 y_vector_class.cc

```
#include "y_vector_class.h"
/******************************************************************************/
yVector::yVector(){
  Resize(0);
}/****************************************************************************/
yVector::yVector(int size){
  Resize(size);
}/****************************************************************************/
yVector::yVector(const string &vec){
  (*this)=vec;
}/****************************************************************************/
int yVector::Size() const{
  return size_;
}/****************************************************************************/
void yVector::Resize(int size){
  store_.resize(size,0);
  size_=size;
}/****************************************************************************/
void yVector::Show() const{
  for(int i=0;i<size_;++i) printf("%f,",store_[i]);
  printf("\b \n");
}/****************************************************************************/
void yVector::Sort(){
  for(int i=1;i<size_;++i) if(store_[i-1]>store_[i]){
    double hold=store_[i];
    store_[i]=store_[i-1];
    store_[i-1]=hold;
    if(i>1) i-=2;
  }
}/*****************************************************************************/
double yVector::yMin() const{
  double min=store_[0];
  for(int i=1;i<size_;++i) if(store_[i]<min) min=store_[i];
  return min;
}/*****************************************************************************/
double yVector::yMax() const{
  double max=store_[0];
  for(int i=1;i<size_;++i) if(store_[i]>max) max=store_[i];
  return max;
}/*****************************************************************************/
double yVector::Avg() const{
  double sum=0;
  for(int i=0;i<size_;++i) sum+=store_[i];
  return sum/double(size_);
}/*****************************************************************************/
yVector yVector::Abs() const{
  yVector abs=*this;
  for(int i=0;i<size_;++i) abs[i]=fabs(abs[i]);
  return abs;
}/*****************************************************************************/
double &yVector::operator[](int i){
  return store_[i];
}/*****************************************************************************/
double yVector::operator[](int i) const{
  return store_[i];
}/*****************************************************************************/
yVector yVector::operator+=(double new_element){
  Resize(size_+1);
  store_[size_-1]=new_element;
  return *this;
}/*****************************************************************************/
yVector yVector::operator=(const string &elements){
  Resize(0);
  for(int n=0,k;n<int(elements.size());++n){
    for(k=0;elements[n]!=','&&n<int(elements.size());++k,++n);
    *this+=atof(elements.substr(n-k,k).c_str());
  }
  return *this;
}/*****************************************************************************/
yVector yVector::operator*(double right) const{
  yVector product=*this;
  for(int i=0;i<Size();++i) product[i]*=right;
  return product;
}/*****************************************************************************/
yVector yVector::operator+(const yVector &right) const{
  if(Size()!=right.Size()) Error("yVector addition error (size mismatch)");
  yVector sum=right;
  for(int i=0;i<Size();++i) sum[i]+=store_[i];
  return sum;
}/*****************************************************************************/
yVector yVector::operator-(const yVector &right) const{
  return (*this)+(right*(-1));
}/*****************************************************************************/
```

B.7 y_matrix_class.h

```
   /************************************************************************
 ******************************************************************************
***                                                                          ***
**                            MATRIX CONTAINER CLASS                          **
**                           version 1.10 (01-23-2011)                        **
**                                  -Rob Yager                                **
***                                                                          ***
 ******************************************************************************
    ************************************************************************
This is a two-dimensional matrix container class.  Matrix elements are accessed
in the same manner as traditional c++ arrays.  Operator overriding has been used
to allow for matrix addition, matrix subtraction, matrix multiplication, and
scalar multiplication using traditional notation.  Matrix transpose and inverse
functions are also included.
*******************************************************************************/
#ifndef Y_MATRIX_CLASS_H_
#define Y_MATRIX_CLASS_H_
/*******************************************************************************/
#include "y_vector_class.h"
/*******************************************************************************/
class yMatrix{
public://---------------------PUBLIC PARAMETERS---------------------------------
  string format_;
  string append_;
  string delimiter_;
  string indent_;
  //---------------------------PUBLIC FUNCTIONS---------------------------------
  void SetDefaults(){//..............assigns default values to member parameters
    format_="%+.6e";
    append_="w";
    delimiter_=",";
    indent_="  ";
  }
  yMatrix();//...................default constructor (size is initially unknown)
  yMatrix(int rows,int cols);//............constructor (size is initially known)
  yMatrix(const string &matrix);//.............constructor (entire matrix known)
  yMatrix(const yVector &first_row);//.............constructor (first row known)
  int Size(int dimension) const;//.......returns the size of the input dimension
  void Resize(int rows,int cols);//.................changes the size of a matrix
  void PushBack(const yVector &new_row);//...adds new row to the end of a matrix
  void Show() const;//.........................displays the contents of a matrix
  yVector Row(int i) const;//....................returns the ith row of a matrix
  yVector Col(int j) const;//.................returns the jth column of a matrix
  yMatrix T() const;//.........................returns the transpose of a matrix
  void Identity(int n);//..............converts a matrix into an identity matrix
  void Zero();//..................converts an existing matrix into a zero matrix
  yMatrix Inverse() const;//.....................returns the inverse of a matrix
  void WriteFile(string filename) const;//.............writes a matrix to a file
  //--------------------------OPERATOR OVERLOADS--------------------------------
  yVector &operator[](int i);//............allows matrix elements to be accessed
  yVector operator[](int i) const;//.......allows matrix elements to be accessed
  yMatrix operator=(const string &elements);//................populates a matrix
  yMatrix operator+(const yMatrix &right) const;//.......the sum of two matrices
  void operator+=(const yVector &new_row);//.............overload for PushBack()
  yMatrix operator-(const yMatrix &right) const;//....diff. between two matrices
  yMatrix operator*(const yMatrix &right) const;//...the product of two matrices
  yMatrix operator*(double scalar) const;//.........product of matrix and scalar
  yMatrix operator*(const yVector &right) const;//..matrix/vector multiplication
  yVector ToVector() const;//....................converts a yMatrix to a yVector
private://--------------------------PRIVATE-------------------------------------
  int rows_;//....................................the number of rows in a matrix
  int cols_;//................................the number of columns in a matrix
  vector <yVector> store_;//....................storage for all matrix elements
  yMatrix LuDcmp(vector <int> &indx) const;//......helper function for inverse()
  yVector LuBkSb(const vector <int> indx, const yVector b) const;//.......helper
};/***************************************************************************/
#endif/*************************************************************************/
```

B.7 y_matrix_class.cc

```
#define _CRT_SECURE_NO_WARNINGS//.................disables deprecation warnings
#include "y_matrix_class.h"
/*****************************************************************************/
yMatrix::yMatrix(){
  Resize(0,0);
  SetDefaults();
}/***************************************************************************/
yMatrix::yMatrix(int rows,int cols){
  Resize(rows,cols);
  SetDefaults();
}/***************************************************************************/
yMatrix::yMatrix(const string &matrix){
  SetDefaults();
  *this=matrix;
}/***************************************************************************/
yMatrix::yMatrix(const yVector &first_row){
  Resize(1,first_row.Size());
  store_[0]=first_row;
  SetDefaults();
}/***************************************************************************/
int yMatrix::Size(int dimension) const{
  if(dimension==0) return rows_;
  if(dimension==1) return cols_;
  else Error("Invalid yMatrix.Size() argument.");
  return 1;
}/***************************************************************************/
void yMatrix::Resize(int rows,int cols){
  for(int i=0;i<(int)store_.size();++i) store_[i].Resize(cols);
  store_.resize(rows,yVector(cols));//...........use the vector resize function
  rows_=rows;
  cols_=cols;
}/***************************************************************************/
void yMatrix::PushBack(const yVector &new_row){
  if(new_row.Size()!=cols_) Error("yVector/yMatrix size mismatch.");
  rows_++;
  store_.push_back(new_row);
}/***************************************************************************/
void yMatrix::Show() const{
  for(int i=0;i<rows_;++i){
    printf(indent_.c_str());
    for(int j=0;j<cols_-1;++j){
      printf(format_.c_str(),store_[i][j]);
      printf(delimiter_.c_str());
    }
    printf(format_.c_str(),store_[i][cols_-1]);
    printf("\n");
  }
}/***************************************************************************/
yVector yMatrix::Row(int i) const{
  return store_[i];
}/***************************************************************************/
yVector yMatrix::Col(int j) const{
  yVector colvector(rows_);
  for(int i=0;i<rows_;++i) colvector[i]=store_[i][j];
  return colvector;
```

```
}/****************************************************************************/
yMatrix yMatrix::T() const{
  yMatrix trans(cols_,rows_);
  for(int i=0;i<rows_;++i) for(int j=0;j<cols_;++j) trans[j][i]=store_[i][j];
  return trans;
}/****************************************************************************/
void yMatrix::Identity(int n){
  Resize(0,0);
  Resize(n,n);
  for(int i=0;i<n;++i) store_[i][i]=1.0;
}/****************************************************************************/
void yMatrix::Zero(){
  for(int i=0;i<rows_;++i) for(int j=0;j<cols_;++j) store_[i][j]=0;
}/****************************************************************************/
yMatrix yMatrix::Inverse() const{
  //Make sure that the matrix is square.
  if(rows_!=cols_) Error("Matrix rows don't equal matrix columns.");
  vector <int> indx(rows_);
  yVector col(rows_);
  yMatrix invrs(rows_,rows_);
  yMatrix LU;
  LU=LuDcmp(indx);
  for(int j=0;j<rows_;++j){
    for(int i=0;i<rows_;++i) col[i]=0.0;
    col[j]=1.0;
    col=LU.LuBkSb(indx,col);
    for(int i=0;i<rows_;++i) invrs.store_[i][j]=col[i];
  }
  return invrs;
}/****************************************************************************/
void yMatrix::WriteFile(string filename) const{
  FILE *stream;
  stream=fopen(filename.c_str(),append_.c_str());
  for(int i=0;i<rows_;++i){
    fprintf(stream,indent_.c_str());
    for(int j=0;j<cols_-1;++j)  {
      fprintf(stream,format_.c_str(),store_[i][j]);
      fprintf(stream,delimiter_.c_str());
    }
    fprintf(stream,format_.c_str(),store_[i][cols_-1]);
    fprintf(stream,"\n");
  }
  fclose(stream);
}/****************************************************************************/
yVector &yMatrix::operator[](int i){
  return  store_[i];
}/****************************************************************************/
yVector yMatrix::operator[](int i) const{
  return  store_[i];
}/****************************************************************************/
yMatrix yMatrix::operator=(const string &elems){
  Resize(0,0);
  for(int i=0,n=0;n<int(elems.size());++i,++n)
    for(int j=0,k;n<int(elems.size())&&elems[n]!=';';++j,++n){
      for(k=0;elems[n]!=','&&elems[n]!=';'&&n<int(elems.size());++k,++n);
      if(j>=cols_) Resize(rows_,j+1);
      if(i>=rows_) Resize(i+1,cols_);
      store_[i][j]=atof(elems.substr(n-k,k).c_str());
      if(elems[n]==';') --n;
    }
  return *this;
}/****************************************************************************/
yMatrix yMatrix::operator+(const yMatrix &right) const{
  //Make sure that the matrices are sized correctly
  if(rows_!=right.rows_||cols_!=right.cols_)Error("yMatrix size mismatch");
  //Perform addition.
  yMatrix sum=right;
  for(int i=0;i<rows_;++i) for(int j=0;j<cols_;++j)
    sum[i][j]+=store_[i][j];
  return sum;
}/*****************************************************************************/
void yMatrix::operator+=(const yVector &new_row) {
  PushBack(new_row);
}/*****************************************************************************/
yMatrix yMatrix::operator-(const yMatrix &right) const{
  return (*this)+(right*(-1.0));
}/*****************************************************************************/
yMatrix yMatrix::operator*(const yMatrix &right) const{
  //Make sure that the matrices are sized correctly.
  if(cols_!=right.rows_) Error("yMatrix size mismatch");
  //Perform multiplication.
  yMatrix product(rows_,right.cols_);
  for(int i=0;i<rows_;++i) for(int j=0;j<right.cols_;++j)
    for(int k=0;k<cols_;++k) product[i][j]+=store_[i][k]*right[k][j];
  return product;
}/*****************************************************************************/
yMatrix yMatrix::operator*(double scalar) const{
  yMatrix product(rows_,cols_);
  for(int i=0;i<rows_;++i) for(int j=0;j<cols_;++j)
    product[i][j]=store_[i][j]*scalar;
  return product;
}/*****************************************************************************/
yMatrix yMatrix::operator*(const yVector &right) const{
  if(Size(1)!=right.Size()) Error("yMatrix Vector Multiplication Error\n");
  yMatrix matrix_right(right.Size(),1);
  for(int i=0;i<right.Size();++i) matrix_right[i][0]=right[i];
  return (*this)*matrix_right;
}/*****************************************************************************/
yVector yMatrix::ToVector() const{
  yVector out;
  if(Size(0)!=1&&Size(1)==1){
    out.Resize(Size(0));
    for(int i=0;i<Size(0);++i) out[i]=store_[i][0];
  }
  if(Size(0)==1&&Size(1)!=1){
    out.Resize(Size(1));
    for(int i=0;i<Size(1);++i) out[i]=store_[0][i];
  }
  if((Size(0)!=1&&Size(1)!=1)||(Size(0)==1&&Size(1)==1))
    Error("Can't convert yMatrix to yVector.");
  return out;
}/*****************************************************************************/
yMatrix yMatrix::LuDcmp(vector <int> &indx) const{
   yMatrix LU=*this;
   const double TINY=1.0e-20;
   int imax;
   yVector vv(rows_);
   //Find the largest element in the matrix
   for(int i=0;i<rows_;++i ){
     double max=LU[i].Abs().yMax();
     if(max==0.0) Error("yMatrix is singular.");
     vv[i]=1.0/max;
   }
   for(int j=0;j<rows_;++j){
      for(int i=0;i<j;++i){
         double sum=LU[i][j];
         for(int k=0;k<i;++k) sum-=LU[i][k]*LU[k][j];
         LU[i][j]=sum;
      }
      double big=0.0,dum;
      for(int i=j;i<rows_;++i ){
         double sum=LU[i][j];
         for(int k=0;k<j;++k) sum-=LU[i][k]*LU[k][j];
         LU[i][j]=sum;
         if((dum=vv[i]*fabs(sum))>=big){
            big=dum;
            imax=i;
         }
      }
      if(j!=imax){
         for(int k=0;k<rows_;++k){
            double dum=LU[imax][k];
            LU[imax][k]=LU.store_[j][k];
            LU[j][k]=dum;
         }
         vv[imax]=vv[j];
      }
      indx[j]=imax;
      if(LU[j][j]==0.0) LU[j][j]=TINY;
      if(j!=rows_-1) for(int i=j+1;i<rows_;++i) LU[i][j]/=LU[j][j];
   }
   return LU;
}/*****************************************************************************/
yVector yMatrix::LuBkSb(vector <int> indx,yVector b) const{
  int ii=0;
  for(int i=0;i<rows_;++i){
    double sum=b[indx[i]];
    b[indx[i]]=b[i];
    if(ii) for(int j=ii-1;j<=i;++j) sum-=store_[i][j]*b[j];
    else if(sum) ii=i+1;
    b[i]=sum;
  }
  for(int i=rows_-1;i>=0;--i){
    double sum=b[i];
    for(int j=i+1;j<cols_;++j) sum-=store_[i][j]*b[j];
    b[i]=sum/store_[i][i];
  }
  return b;
}/*****************************************************************************/
```

NO. OF COPIES	ORGANIZATION
1 (PDF only)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA 8725 JOHN J KINGMAN RD STE 0944 FORT BELVOIR VA 22060-6218
1	DIRECTOR US ARMY RESEARCH LAB IMNE ALC HRR 2800 POWDER MILL RD ADELPHI MD 20783-1197
1	DIRECTOR US ARMY RESEARCH LAB RDRL CIM L 2800 POWDER MILL RD ADELPHI MD 20783-1197
1	DIRECTOR US ARMY RESEARCH LAB RDRL CIM P 2800 POWDER MILL RD ADELPHI MD 20783-1197
1	DIRECTOR US ARMY RESEARCH LAB RDRL D 2800 POWDER MILL RD ADELPHI MD 20783-1197
	ABERDEEN PROVING GROUND
1	DIR USARL RDRL CIM G (BLDG 4600)

NO. OF COPIES	ORGANIZATION
	ABERDEEN PROVING GROUND
25	DIR USARL
(24 HC	RDRL WM
1 CD)	P PLOSTINS
	RDRL WML
	M ZOLTOSKI
	J NEWILL
	E SCHMIDT
	RDRL WML A
	M ARTHUR
	B BREECH
	B FLANDERS
	W OBERLE
	C PATTERSON
	R PEARSON
	L STROHM
	P WYANT
	R YAGER (3 HC, 1 CD)
	RDRL WML B
	J MORRIS (A)
	RDRL WML C
	K MCNESBY (A)
	RDRL WML D
	R BEYER (A)
	RDRL WML E
	P WEINACHT
	RDRL WML F
	D LYON
	RDRL WML G
	T GORDON BROWN (A)
	RDRL WML H
	C CANDLAND (A)
	RDRL WMM
	J ZABINSKI (A)
	RDRL WMP
	P BAKER

INTENTIONALLY LEFT BLANK.